Thursday | 4:37 PM
CLASSTIME Pg. No.
Date 08/10/2022

OOPS OOPS OOPS

Oop's Concept :-

(1) Data hiding
(2) Abstraction
(3) Encapsulation
(4) Tightly encapsulated class
(5) IS-A Relationship
(6) HAS-A Relationship (MCU)
(7) Method Signature
(8) Overloading
(9) Overriding
(10) Static Control flow
(11) Instance Control flow
(12) Constructors - 4
(13) Coupling
(14) Cohesion
(15) Type-Casting

1 module

theoretical concepts

talks about only Security

2.2.5 1.7 3.5

framework related things.

(1) Data hiding :- Outside person can't access our internal data directly. Or our internal data should not go out directly. This OOP feature is nothing but data hiding..

After validation/Authentication outside person can access our internal data.

for eg :- (1) After providing proper username & password we can able to access our

main inbox information.

eg2: even though we are valid customer of the bank we can able to access our account infom. and we can't access others account information.

By declaring data member as (variable) as private we can achieve data hiding.

eg: Public class Account
{
    private double balance;
    :
    :
    Public double getBalance()
    {
        validation
        return balance;
    }
    :
}

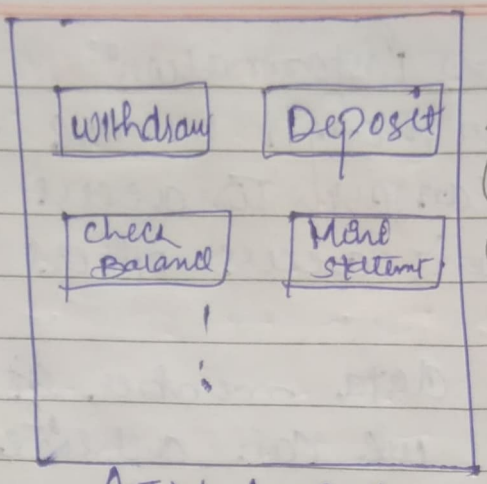
* The main advantage of data hiding is security.

Note: It is highly recommended to declare data member (variable) as private.

(2) Abstraction: Hiding internal implementation and just highlight the set of services what we are offering is a concept of Abstraction.

eg: Through Bank ATM GUI Screen Bank people are highlighting a set of services what they are offering without highlighting internal imple

ment".



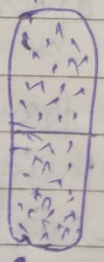
ATM GUI Screen.

The main advantages of abstraction are
(1) Security - we can achieve security bcz we are not highlighting our internal implementn.
(2) Without affecting outside person we can able to perform any type of changes in our internal system, hence enhancement will become easy.
(3) It improve maintainability of the appli
(4) It improves easyness to use our system.

(5) By using interface and abstract classes, we can implement abstraction.

(3) Encapsulation :- The process of binding data and corresponding methods into a single unit called Encapsulation.

eg:
1. Class Student
{
data members
+
methods (behaviour)
}

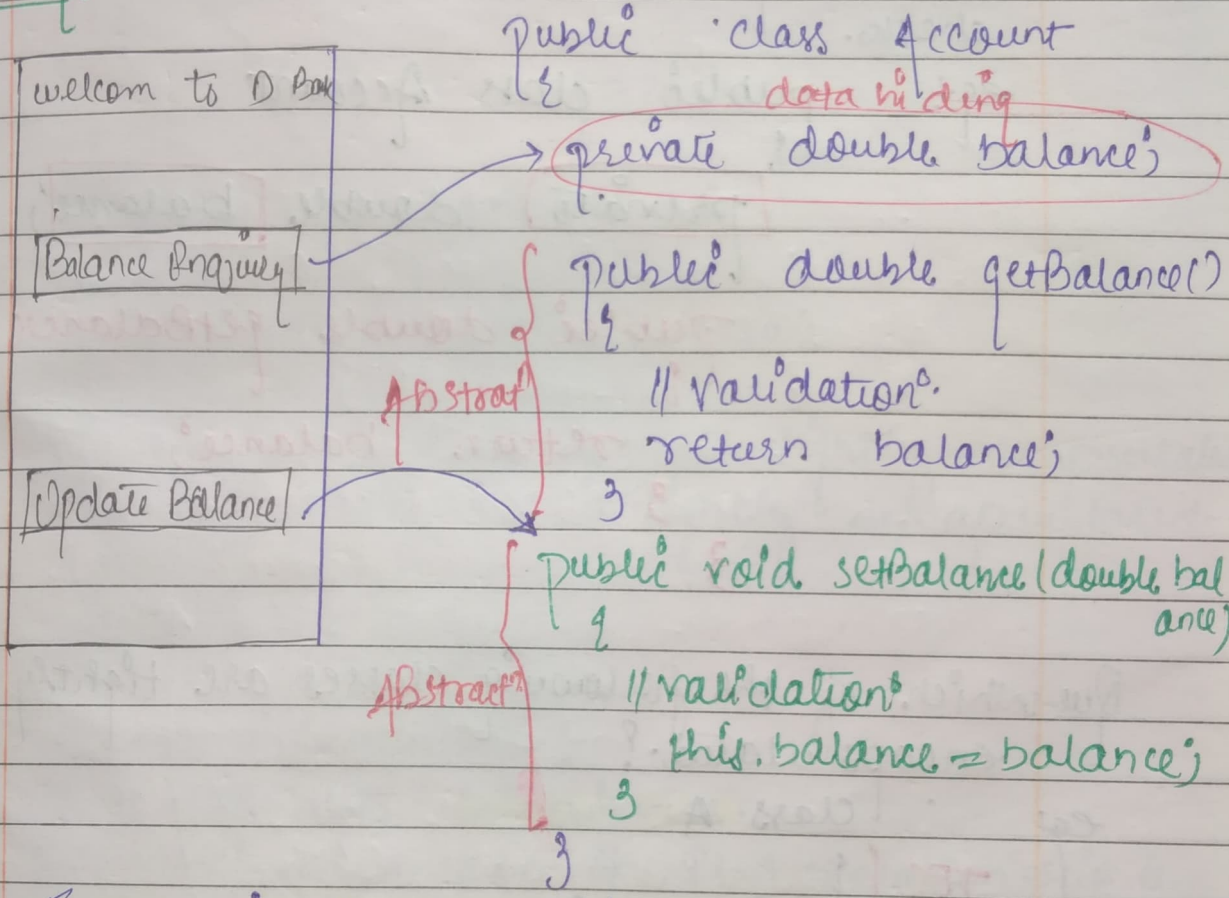


Capsule

Encapsulation = Data hiding + Abstraction.

- If any compont follows Data hiding & abstraction such type of compoment is said to be encapsulated Compoment.

eg:



- The main advantages of encapsulation are -
(i) we can acheive security.
(ii) Enhancement will become easy.
(iii) It improves maintainability of the applicn.
(iv)

* The main advantge of encapsulation is we can acheive security but the main disadvantage of encapsulation is it increases length of the code & slows down execution.

CLASSTIME Pg. No.
Date / /

(4) Tightly Encapsulated class :- A class is said to be tightly encapsulated iff each & every variable declared as private.
whether class contains corresponding getter and setter methods or not and whether these methods are declared as public or not these things we are not required to check.

eg:-
```
public class Account
{
    private double balance;

    public double getBalance()
    {
        return balance;
    }
}
```

Que- Which of the following classes are tightly encapsulated.?

eg:-
```
class A
{
    private int x=10;
}

class B extends A
{
    int y=20;
}

class C extends A
{
    private int z=30;
}
```

TEC

No

TEC

Q- Which of the following classes are tightly encapsulated.?

```
class A
{
    int x=10;
}
```

```
class B extends A
{
    private int y=20;
}
```

```
class C extends B
{
    private int z=30;
}
```

If the parent class is not tightly encapsulated then no child class is tightly encapsulated.

* Lecture 52 :-

(5) Is-A Relationship :-

(i) It is also known as inheritance.

(ii) The main advantage of Is-A relationship is Code reusability.

(iii) By using extends keywords we can implement is a relationship.

```
class P
{
    public void m1()
    {
        System.out.println("parent");
    }
```

```
}
class C extends P
{
    public void m2()
    {
        sopln("child");
    }
}

class Test
{
    p s v main(String[] args)
    {
```

① `P p = new P();`
`p.m1();` ✓
`p.m2();` ✗ → CE: Cannot find symbol
symbol: method m2()
location: class P.

② `C c = new C();`
`c.m1();` ✓
`c.m2();` ✓

③ `P p1 = new C();`
`p1.m1();` ✓
`p1.m2();` ✓

④ `C c1 = new P();` → CE: incompatible types
found: P
required: C

```
    }
}
```

Conclusions:-

(1) Whatever methods parent has by default available to the child. and hence on the child reference we can call both parent class & child class methods.

(2) Whatever methods child has by default not available to the parent and hence on the parent reference we can't call child specific methods.

(3) Parent reference can be used to hold child object, but by using that reference we can't call child specific methods. but we can call the methods present in parent class.

(4) Parent reference can be used to hold child object, but child reference can't be used to hold parent object.

Without Inheritance:-

Class Vloan	Class Hloan	Class Ploan
{	{	{
300 methods	300 methods	300 methods
}	}	}

{ 900 method
50 lrs. }

CLASSTIME Pg. No.
Date / /

With Inheritance.

```
class Loan
{
   250 common methods.
}

class Vloan extends Loan
{
   50 specific methods
}

Class Hloan extends loan
{
   50 specfc methds
}

Class Ploan extends loan
{
   50 specfc methods
}
```

{ 400 methods
  40 hours. }

Note: The most common methods which are applicable for any type of child, we have to define in parent class.

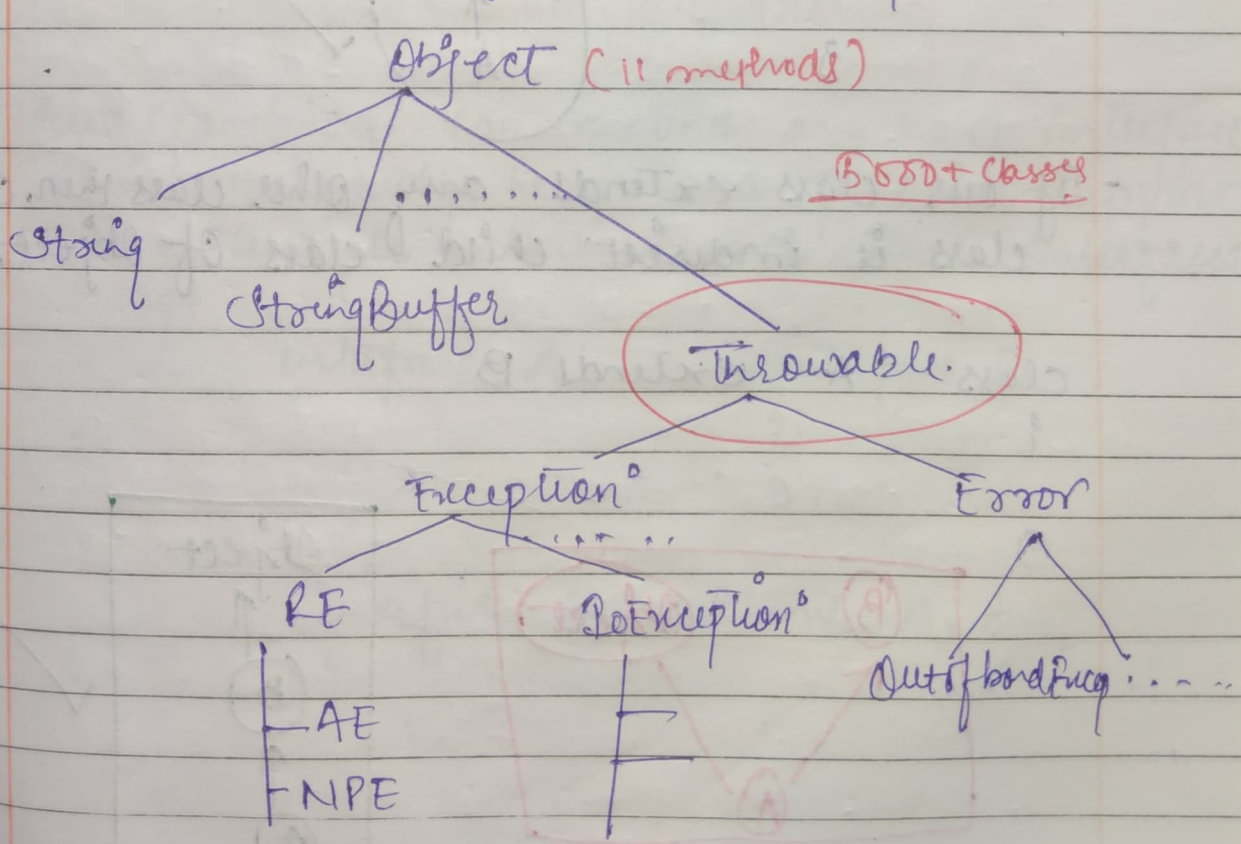
The specific methods which are applicable for a particular child we have to define in child class.

CLASSTIME Pg. No.
Date / /

Total Java API is implemented based on inheritance concept.

→ The most common methods which are applicable for any Java object are defined in Object class. and hence every class in Java is a child class of Object. either directly or indirectly. so that Object class methods by default available to every Java class without rewriting due to this Object class act as root for all Java classes.

→ Throwable class defines the most common methods which are required for every exception and error classes hence this class act as root for Java exception hirarchy



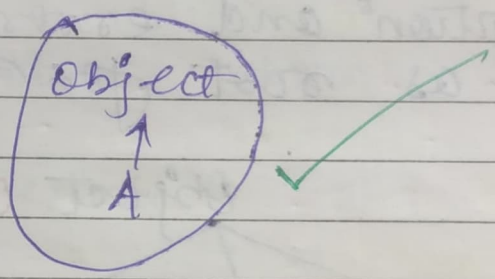
CLASSTIME Pg. No.
Date / /

Multiple Inheritance: A Java class can't extend more than one class at a time. Hence Java won't provide support for multiple inheritance in classes.

eg:-
```
class A extends B, C
{
}
```
CE:

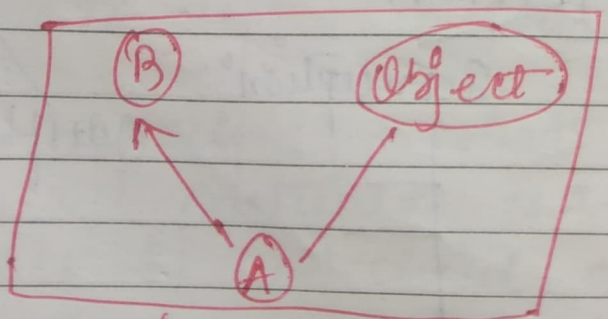
Note: If our class doesn't extend any other class then only our class is direct child class of Object.

eg:-
```
class A
{
}
```

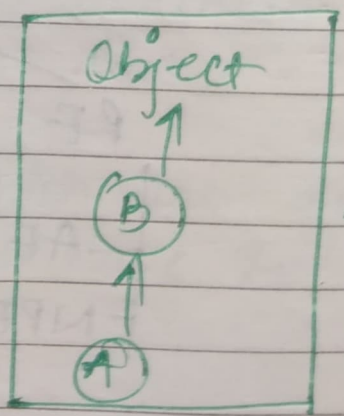


- If our class extends any other class then our class is indirect child class of Object.

```
class A extends B
{
}
```



Multiple Inheritance



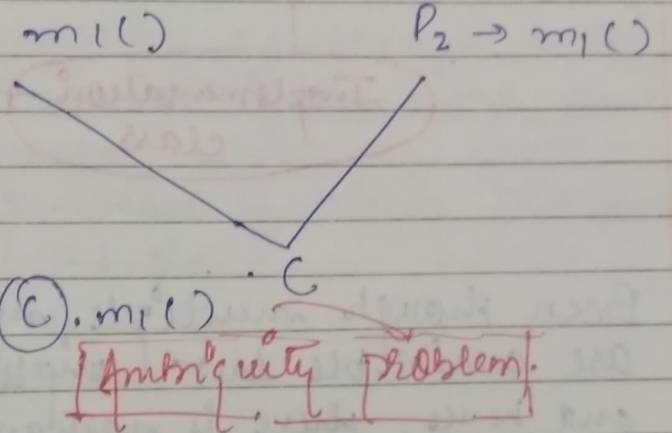
Multilevel inheritance

CLASSTIME Pg. No.
Date / /

Note: Either directly or indirectly Java won't provide support for inheritance w.r.t classes.

Q- Why Java won't provide support for multiple inheritance?

There may be a chance of Ambiguity problem, hence Java won't provide support for multiple inheritance.



But interface can extend any no of interface simultaneously, hence Java provide support for multiple inheritance w.r.t interfaces.

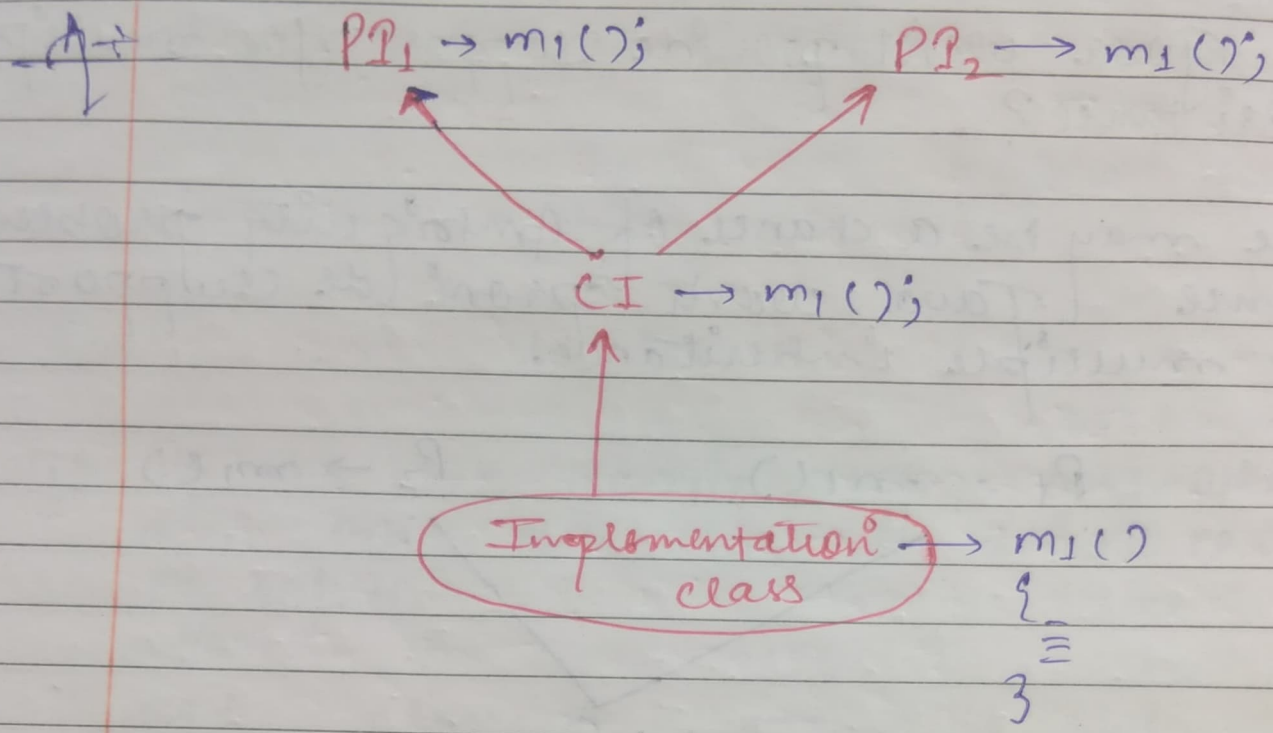
eg:

```
interface A
{
}

interface B
{
}

interface C extends A, B
{
}
```

Q Why ambiguity problem won't be there in interfaces??



Even though multiple method declarations are available but implementation is unique. and hence, there is no chance of ambiguity problem in interfaces.

Note: Strictly speaking through interfaces we don't get inheritance.

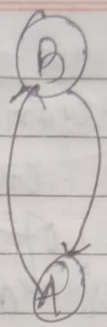
* Loophole:
Cyclic inheritance: Cyclic inheritance is not allowed in java. Of course it is not required.

eg: class A extend A
{
}

CE: cyclic inheritance involving A.

CLASSTIME Pg. No.
Date / /

```
class A extends B
{
}
class B extends A
{
}
```



CE! cyclic inheritance involving A.

(6) Has-A Relationship :- (MCU)

(i) Has-A Relationship is also Has Composition or Aggregation.

(ii) There is no specific keyword to implement has-A relation but most of the time we are depend on new keyword.

(iii) The main advantage of Has-A relationship is Reusability of the code.

eg:-

```
class Car
{
    Engine e = new Engine();
}
```

Car Has-A Engine Reference

```
class Engine
{
    // Engine specific functionality
}
```

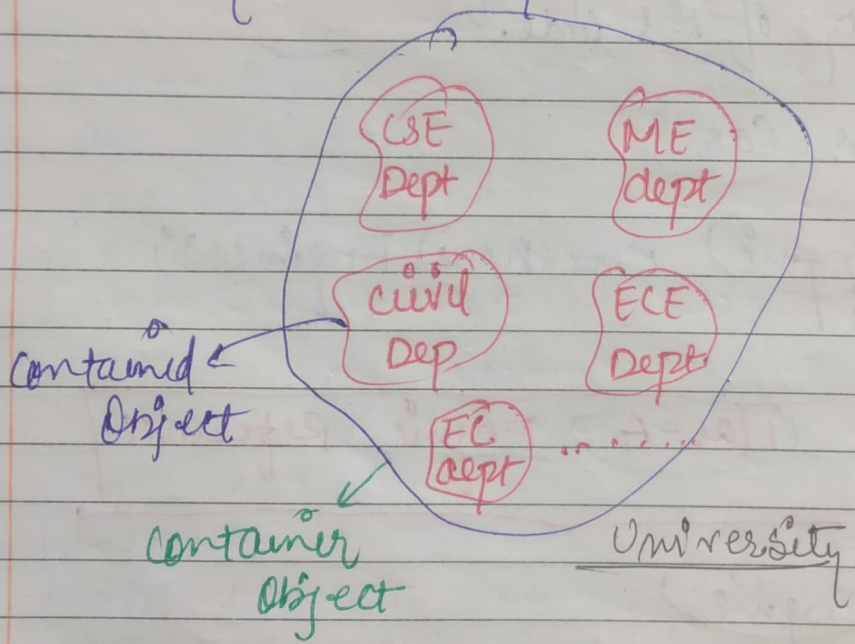
CLASSTIME Pg. No.
Date 12/08/2022

* Difference b/w Composition and Aggregation:

(1) * Composition :-

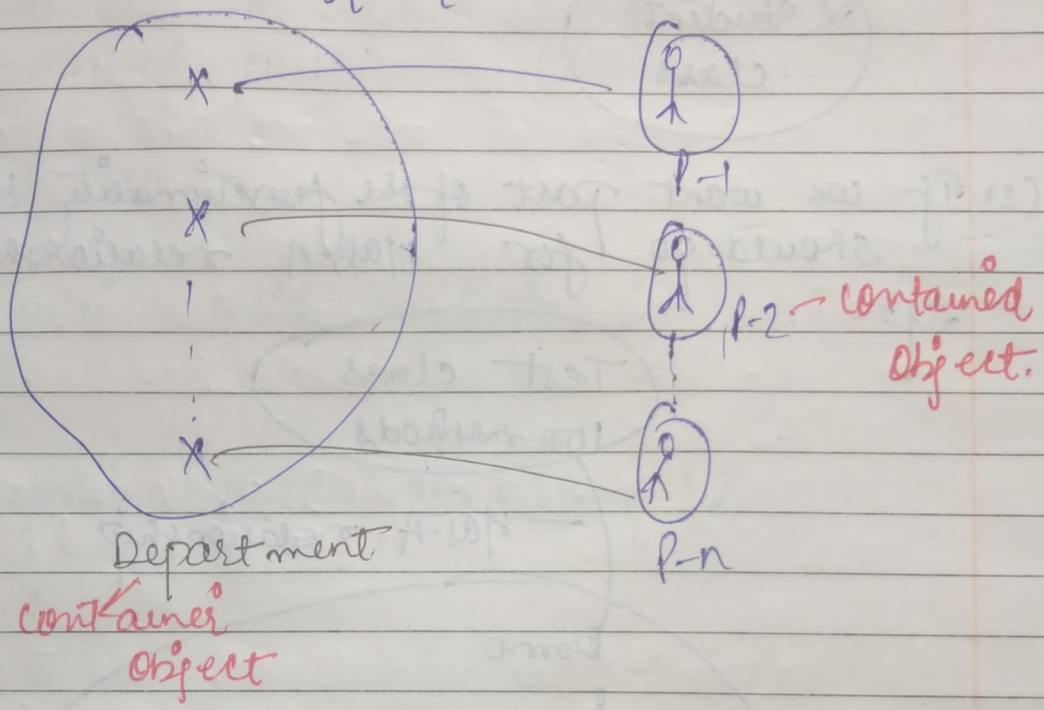
(1) Without existing Container object, if there is no chance of existing contained objects, then Container and Contained objects are strongly associated, and this strong association is nothing but Composition.

eg:- University consists of several departments without existing University, there is no chance of existing department. Hence University and departments are strongly associated and this strong association is nothing but Composition.



(2) Aggregation - Without existing Container object there is a chance of existing Contained object. then Container and Contained Objects are weakly associated. and this weak associat^n is nothing but aggregation.

eg: Department consists of several professors. without existing department there may be a chance of existing professor objects. hence deptmnt and professor objects are weakly associated and this weak association is nothing but aggregation.



Note: In composition objects are strongly associated. whereas, in aggregation objects are weakly associated.

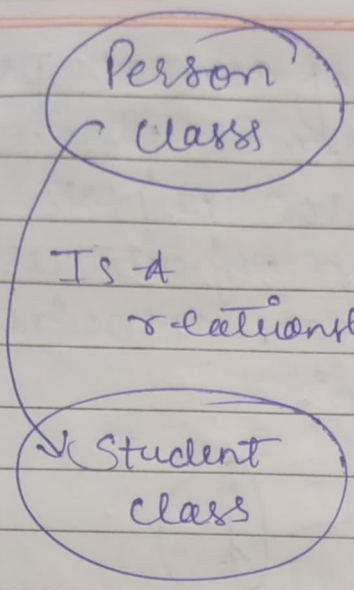
In composition container objects holds directly contained objects.

whereas, in aggregation container objects holds just references of contained objects.

(i) IS-A vs HAS-A

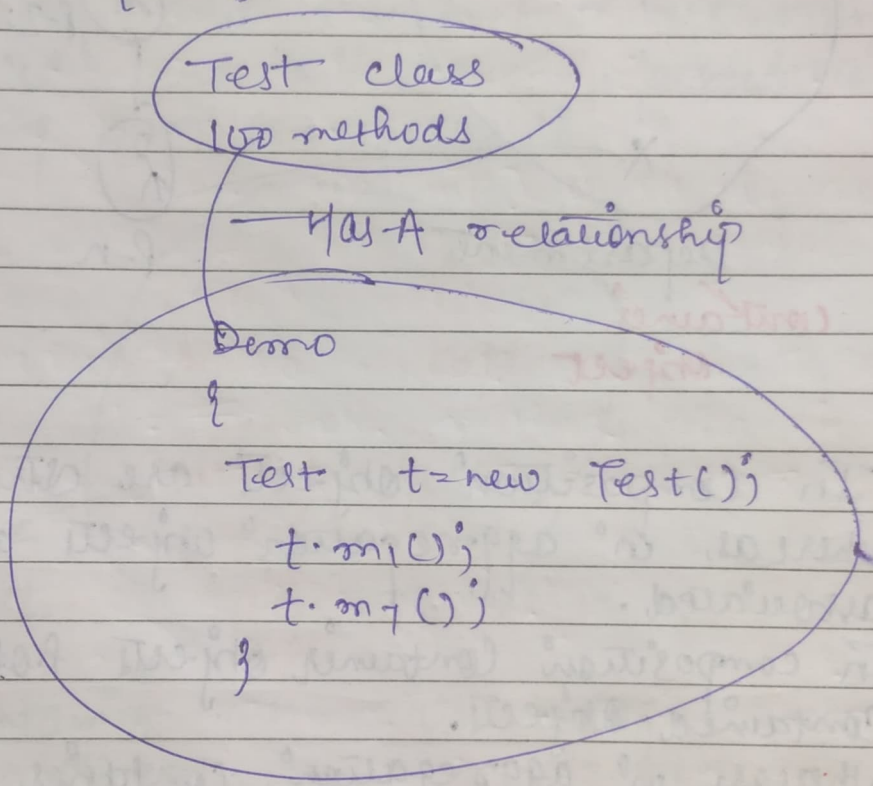
If we want total functionality of a class, automatically then we should go for IS-A relationship.

eg:



(2) If we want part of the functionality, then we should go for Has-A relationship.

eg:-



(7) Method Signature :- In Java method signature consist of method names followed by argument types.

eg:- public static int m1(int i, float f)

AM RT MN AT

CLASSTIME Pg. No.
Date / /

m1 (int, float) → method signature.

⇒ * Return type is not part of method signature in Java.

* Compiler will use method signature to resolve ~~to~~ method calls.

Eg:
```
class Test
{
    public void m1(int i)
    {
    }
    public void m2(String s)
    {
    }
}
```

```
Test t = new Test();
t.m1(10);
t.m2("dunga");
t.m3(10.5);
```
→ CE: Cannot find symbol
Symbol: method m3(double)
location: class Test

class Test

m1(int)
m2(String)

Method Table

Conclusion Req. Medt Sig :- With in a class two methods with the same signature not allowed.

eg:-

```
class Test
{
    public void m1(int i) ≡ m1(int)
    {
    }
    public int m1(int x) ≡ m1(int)
    {
        return 10;
    }
}
Test t=new Test();
t.m1(10);
```

CE: m1(int) is already defined in Test

method sig

(8) Overloading :- Two methods are said to be overloaded iff: ✱ both methods having same name but different argument types.

eg:- m1(int i)
m1(String s)

In C lang. method overloading concept is not available hence we can't declare multiple methods with same name. but different argument types.

If there is change in argument type compulsory we should go for new method name. which increases complexity of programming.

abs (int i) ⇒ abs(10);
labs (long l) ⇒ labs(10l);
fabs (float f) ⇒ fabs(10.5f);

⇒ But in java we can declare multiple methods with same name but different argument types. Such types of methods are called overloaded methods.

eg:- abs (int i)
abs (long l)
abs (float f)
} overloaded methods.

* Having Overloading concept in java reduces complexity of programming.

Eg:-
```
class Test
{
    public void m1()
    {
        sopln("no-arg");
    }
    public void m1(int i)
    {
        sopln("int-arg");
    }
    public void m1(double d)
    {
        sopln("double arg");
    }
}
```
Overloaded methods

```
p s v main(String[] args)
{
    Test t = new Test();
    t.m1();
    t.m1(10);
    t.m1(10.5);
}
}
```

reference type — Test — t = new — Test() — object

t.m1(); — no-arg

t.m1(10); — int-arg

t.m1(10.5); — double-arg.

* In overloading method resolution always taken care by Compiler based on reference type. Hence overloading is also considered as Compile time polymorphism or static polymorphism, or Early binding.

Lecture 54 :- Overloading [Continue >>>]

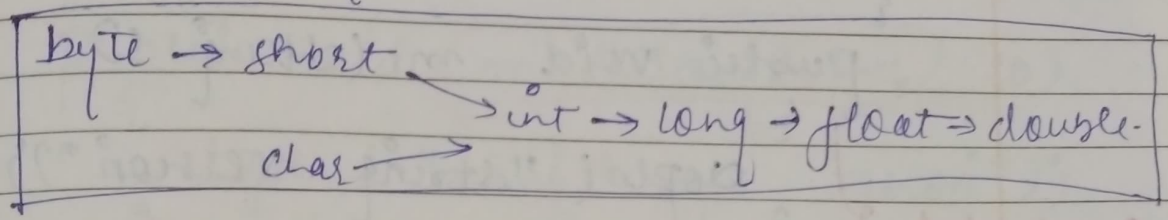
Loopholes

Case 1 :- Automatic promotion in overloading :-

While resolving overloaded methods, if exact match method is not available, then we won't get any CTE immediately. first it will promote argument to the next level and check whether matched method is available or not.

If matched method is available it will be considered. and if the matched method is not available then compiler promote argument once again to the next level

this process will be continued until all possible promotions. Still if the matched method is not available then we will get CTE.
The following are all possible promotions in overloading.



This process is called Automatic promotion in overloading.

Eg: class Test

Overloaded methods.

```
{
    public void m1(int i)
    {
        System.out.println("int-arg");
    }
    public void m1(float f)
    {
        System.out.println("float-arg");
    }
    p s v main(String[] args)
    {
        Test t=new Test();
        t.m1(10);      → int-arg
        t.m1(10.5f);   → float-arg
        t.m1('a');     → int-arg
        t.m1(10l);     → float-arg.
```

CLASSTIME Pg. No.
Date / /

```
        t.m1(10.5);  → CE: cannot find symbol
    }                  symbol: method m1(double)
}                      location: class Test
```

Case2:-

```
class Test
{
    public void m1(String s)
    {
        sopln("String version");
    }
    public void m1(Object o)
    {
        sopln("object version");
    }
    psvm(String[] args)
    {
        Test t=new Test();
        t.m1(new Object); object version
        t.m1("durga"); String version
        t.m1(null); String version
    }
}
```

overloaded methods

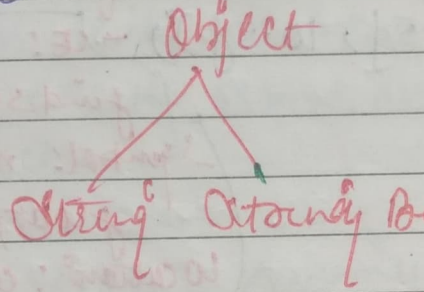
Object
↑
String

NOTE:- while resolving overloaded methods compiler will always give the precedence for child class type argument, when compared with parent type argument.

Case 3 :-

```
class Test
{
    public void m1(String s)
    {
        sopln("String version");
    }
    public void m1(StringBuffer sb)
    {
        sopln("StringBuffer version");
    }
}
```

Overloaded method




```
p s v main(String[] args)
{
    Test t = new Test();
    t.m1("durga");      String version
    t.m1(new StringBuffer("durga"));
                        StringBuffer version
    t.m1(null);  CE: reference to m1() is ambiguous.
}
```

Case 4 :-

```
class Test
{
    public void m1(int i, float f)
    {
        sopln("int-float version");
    }
}
```

CLASSTIME Pg. No.
Date / /

overloaded methods

```
public void m1(float f, int i)
{
    sopln("float-int version");
}

p s v main(String[] args)
{
    Test t = new Test();
    t.m1(10, 10.5f); -int-float version
    t.m1(10.5f, 10); -float-int version
    t.m1(10,10); → CE: reference to m1() is amb
    t.m1(10.5f, 10.5f); → CE: Cannot find symbol.
                          Symbol: method m1(float, float)
                          location: class Test
}
```

Case 5:

overloaded method

```
class Test
{
    public void m1(int x)
    {
        sopln("general method");
    }
    public void m1(int... x)
    {
        sopln("var-arg method");
    }
}
```

```
p s v main(String[] args)
{
    Test t=new Test();
    t.m1();          var-arg method
    t.m1(10,20);     var-arg method
    t.m1(10);        General method.
}
}
```

Conclusion:

* In general var-arg method will get least priority. i.e. if no other method matched then only var-arg method will get the chance. It is exactly same as default case inside switch.

Case 6:

```
class Animal
{
}
class Monkey extends Animal
{
}

class Test
{
public void m1(Animal a)
{
    sopln("Animal version");
}
Public void m1(Monkey m)
{
    sopln("Monkey version");
}
}
```

overloaded methods

CLASSTIME Pg. No.
Date / /

```
P.S.V.M(String[] args)
    Test t = new Test();
(1) Animal a = new Animal();
    t.m1(a); Animal version
(2) Monkey m = new Monkey();
    t.m1(m); Monkey version
(3) Animal a1 = new Monkey();
    t.m1(a1); Animal version.
  }
}
```

Note:- In overloading method resolution always takes care by Compiler, based on reference type. and In overloading runtime object won't play any role.

Lecture 55:- Overriding :-

Whatever methods parent has by default available to the child through inheritance. if child class not satisfied with parent class implementn, then child is allowed to redefine that method based on its requirement. this process is called Overriding.

The parent class method which is overridden is called overridden method. and child class method which is overriding is called

overriding method.

Eg:

```
class P
{
    public void property()
    {
        sopln("cash + land + gold");
    }
    public void marry()
    {
        sopln("Sub Lanna");
    }
}
class C extends P
{
    public void marry()
    {
        sopln("hm | xyz | abc");
    }
}
```

Overridden method.

Overriding

Overriding method

→

```
class Test
{
    p s v main(String[] args)
    {
        ① P p=new P();
           p.marry();  → Parent method
        ② C c=new C();
           c.marry();  → child method
        ③ P p1=new C();
    }
}
```

P.s.marry()' → child method

}

}

* In overridding method resolution always takes care by JVM based on runtime object and hence overriding is also considered as Runtime polymorphism or dynamic polymorphism or Late binding.

* Rules for overriding:

(i) In overriding method names and argument types must be matched. i.e. method signature must become.

(ii) In overriding return types must be same, but this rule is applicable Until 1.4 V only. from 1.5V onwards we can take co-variant return types. Acc. to this child class method return type need not be same as parent method return type. its child types also allowed.

```
class P
{
  public object m1()
  {
    return null;
  }
}
class C extends P
{ public String m1()
```

```
}
return null;
}
}
```

It is invalid in 1.4v version
But from 1.5v onwards it is valid.

Parent class method
Return type

Object

Numeric

Child class method
Return type

Object / String / StringBuffer ... ✓

Number / Integer ..... ✓

String

Object ✗

double

int ✗

Co-variant return type concept applicable only for object-types but not for primitive types.

Parent class private methods not available to the child. Hence overriding concept not applicable for private methods.

CLASSTIME Pg. No.
Date / /

Based on our requirement we can define exactly same private method in child class it is valid but not overriding.

Eg:

```
class P
{
    private void m1()
    {
    }
}
class C extends P
{
    private void m1()
    {
    }
}
```

It is valid but not overriding

We can't override parent class final methods in child classes. If we are trying override we will get CTE

```
class P
{
    public final void m1()
    {
    }
}
class C extends P
{
    public void m1()
}
```

CLASSTIME Pg. No.
Date / /

If m1() in C cannot override in m() in P overriden method is fine

* Parent class abstract methods we should override in child class To provide implementation.

```
abstract class P
{
   public abstract void m1();
}
class C extends P
{
    public void m1()
    {
    }
}
```

We can override non-abstract method as abstract.

```
class P
{
   public void m1()
   {
   }
}
abstract class C extends P
{
   public abstract void m1();
}
```

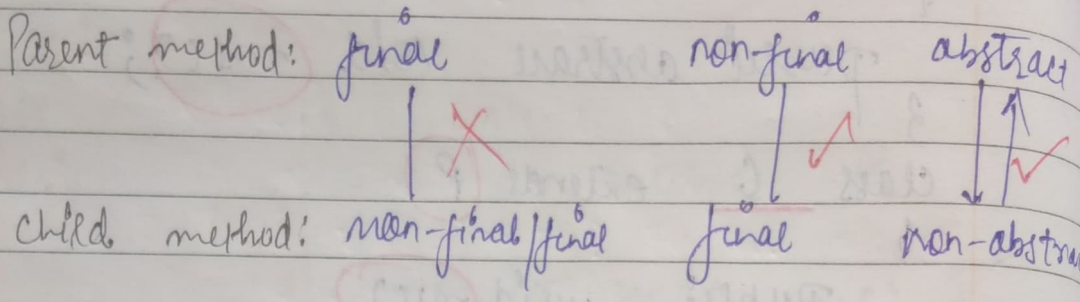
The main advantage of this approach is the availability of parent method

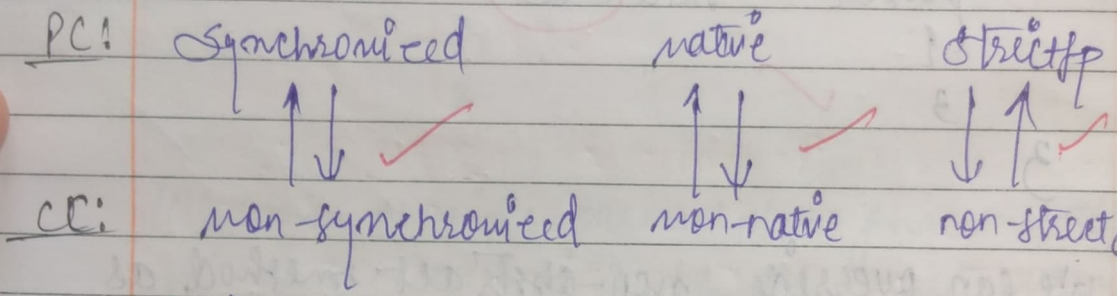
CLASSTIME Pg. No.
Date / /

implementn to the next level child classes.

In overriding the following modifiers won't keep any restriction.

1- Synchronized
2- Native
3- strictfp
4-





→ While overriding we can't reduce scope of access modifier. but we can increase the scope.

```
class P
{
  public void m1()
  {
  }
}
class C extends P
{
  void m1()
  {
  }
}
```

CE: m1() in C cannot override m1() in P; attempting to assign weaker access privileges; was public.

private < default < protected < public

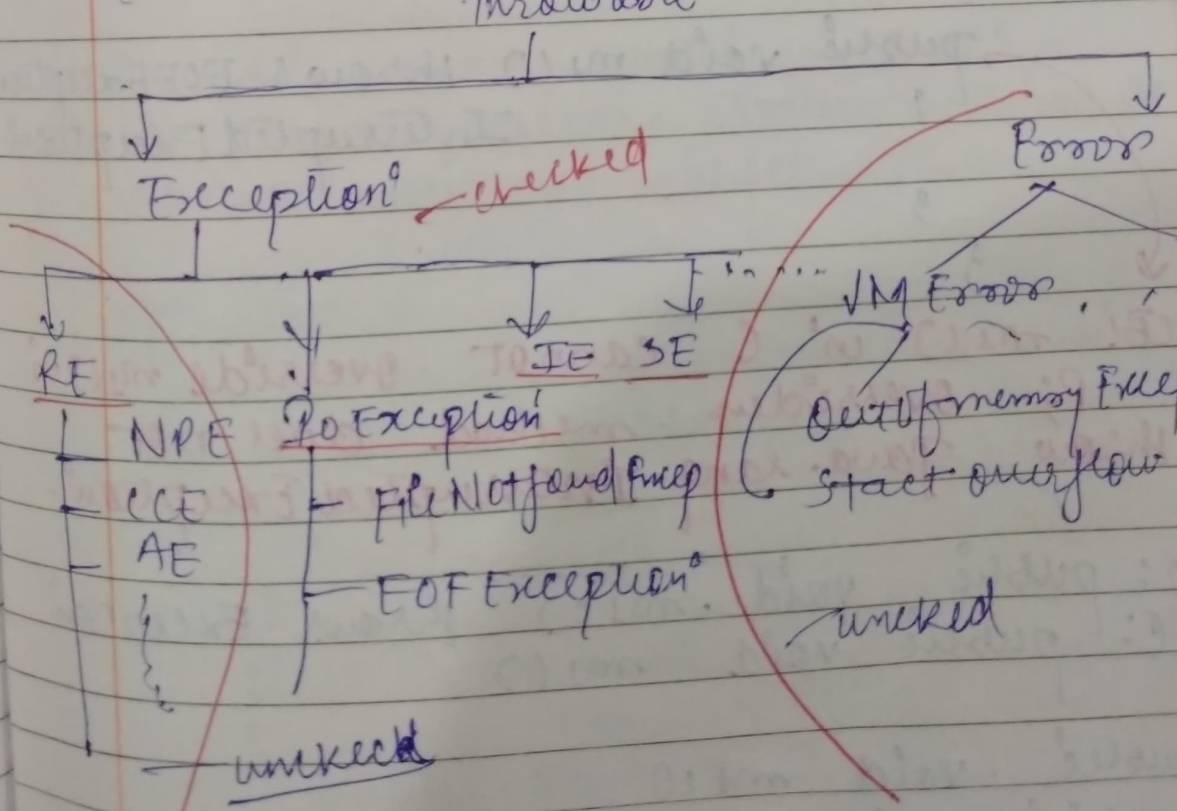
parent class method: public → protected

child class method: public → protected / public

< default > → < default > / protected / public

private → overriding concept not applicable for private methods.

* Lecture 56 & (Session 6 of OOPs)



Rule: If child class method throws any checked exception compulsory parent class method should throw the same checked exception or its parents, otherwise we will get CTE But there are no restrictions for unchecked exceptions.

eg:

```
import java.io.*;
class P
{
public void m1() throws IOException
{
}
}
class C extends P
{
public void m1() throws EOFException,
                        InterruptedException
{
}
}
```

CE: m1() in C cannot override m1() in P; overridden method does not throw java.lang.InterruptedException.

① P: public void m1() throws Exception
C: public void m1()

② P: public void m1()
C: public void m1() throws Exception.

CLASSTIME Pg. No.
Date / /

③ ✓ P: public void m1() throws Exception.
C: public void m1() throws IOException.

④ ✗ P: public void m1() throws IOException
C: public void m1() throws Exception.

⑤ ✓ P: public void m1() throws IOException.
C: public void m1() throws FileNotFoundException, EOFException.

⑥ ✗ P: public void m1() throws IOException.
C: public void m1() throws EOFException, InterruptedException.

⑦ ✓ P: public void m1() throws IOException.
C: public void m1() throws AE, NPE, CCE.

* Overriding wrt static methods:-

(i) We can't override static method as non-static, otherwise we will get CTE.

eg:- class P
{
P s v m1()
{
}
}
class C extends P
{
public void m1()
{
}
}

CE: m1() in C cannot override m1() in P; overridden method is static.

Similarly we can't override a non-static method as static.

eg:
```
class P
{
    public void m1()
    {
    }
}
class C extends P
{
    public static void m1()
    {
    }
}
```

CE: m1() in C cannot override m1() in P; overriding method is static.

If both parent and child class methods are static then we won't get any CTE.
It seems overriding concept is applicable for static methods but it is not overriding and it is method hiding.

```
class P
{
    public static void m1()
    {
    }
}
```

CLASSTIME Pg. No.
Date / /

If a method hiding but not overriding

```
class C extends P
{
    public static void m1()
    {
    }
}
```

*Method Hiding :- All rules of method hiding are exactly same as overriding, except the following differences.

Method Hiding	Overriding
1) Both parent & child class method should be static	1) Both parent & child class method should be non-static
2) Compiler is responsible for method resolution based on reference type.	2) JVM is always responsible for method resolution based on runtime Object.
3) It is also known as Compile Time polymorphism or static polymorphism or early binding.	3) It is also known as Runtime polymorphism or Dynamic polymorphism or Late binding.

eg:-

```
class P
{
    public static void m1()
    {
        sopln("parent");
    }
}
```

```
class C extends P
{
  public static void m1()
  {
    sopln("child");
  }
}

class Test
{
  p s v main(Str...)
  {
    P p = new P();
    p.m1();  → Parent

    C c = new C();
    c.m1();  → child

    P p1 = new C();
    p1.m1();  → ~~child~~ Parent
  }
}
```

Hiding but not overriding.

If both parent & child class methods are non-static then it will become overriding in this case output is: { Parent, child, child. }

Literal Meaning:

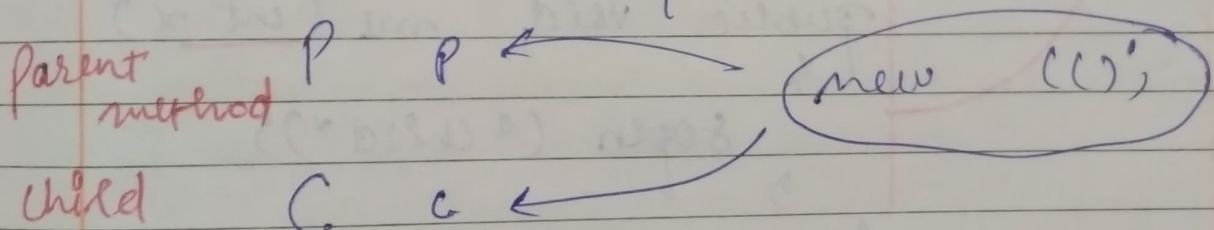
Overriding: In overriding, automatically old copy / parent copy not available / gone only

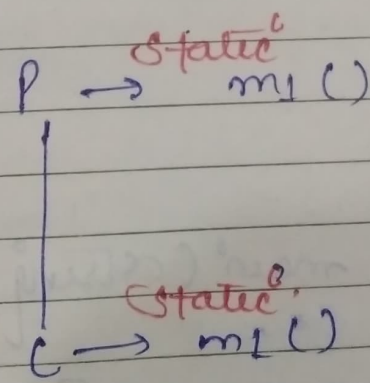
CLASSTIME | Pg. No.
Date / /

child copy / new copy is available.
eg:- for the child object whether we use parent reference or child reference in both cases child class only / p child method only will come in picture.

```
P   P=new C();
    P.m1();  → child method
C   C=new C();
    C.m1();  → child method.
```

Method Hiding :- In Method Hiding whether we use parent or child reference for the child object both method from parent & child will come in picture.





* Overriding w.r.t Var-arg methods :-

Lecture 57 :- >>

We can override var-arg method with another var-arg method only. If we are trying to override with normal method, then it will become overloading but not overriding.

eg:

```
class P
{
    public void m1(int... x)
    {
        sopln("Parent");
    }
}
class C extends P
{
    public void m1(int x)
    {
        sopln("child");
    }
}
```

It is overloading But not overriding

```
class Test
{
    p s v main(String[] args)
    {
        P p = new P();
        p.m1(10); → Parent

        C c = new C();
        c.m1(10); → child
    }
}
```

P p1=new C();
p1.m1(10); child Parent.
3
3

* Overriding w.r.t variables :-
Variable resolution always takes care by Compiler, based on reference type, irrespective of whether the variable is static or non-static. (Overriding concept applicable only for methods but not variables).

eg:- class P
{
int x=888;
}
class C extends P
{
int x=999;
}

class Test
{
P s v main (String[] args)
{
P p=new P();
Sopln(p.x); -888

C c=new C();
Sopln(c.x); -999

P p1=new C();

super.(p1.x); — DDD

3 3

3

P → non-static C → non-static	P → static C → non-static	P → non-static C → static	P → static C → static
DDD	DDD	DDD	DDD
999	999	999	999
BBB	BBB	BBB	BBB

Q- Differences B/w Overloading & Overriding :-

Property	Overloading	Overriding
(1) Method Names -	Must be same ✓	Must be same.
(2) Argument Types	Must be different ✓ [atleast order]	Must be same [including order].
(3) Method Signatures	Must be different	Must be same.
(4) Return Types.	No Restrictions	Must be same until 1.4 From 1.5v onwards co-varient return Types also allowed.
(5) private, static, final.	Can be overloaded	(5) Can not be overridden We can't reduce the scope of access modifier but we can increase the scope.
(6) Access Modifiers.	No Restrictions	

CLASSTIME Pg. No.
Date / /

(7)	Throw clause/ Keywords/ Statement.	No Restrictions	If child class method throws any checked exception compulsory parent class method should through the same checked exception or its parent but no restrictions for unchecked exceptions.
(8)	Method Resolution.	Always takes care by Compiler based on reference type.	(8) Always takes care by JVM based on Runtime Object.
(9)	It is also known as-	Compile Time polym or static polymp. early binding	(9) Runtime polymp. or dynamic polymorph. Late Binding.

Note: In overloading we have to check only method names. [must be same] and arguments Types [must be different].

We are not required to check remaining like return types, access modifiers etc..

But in overriding everything we have to check, like method names argument Types, Return types, access modifiers, throws clause etc.

Qe:- Consider the following method in parent class.?

```
public void m1(int x) throws IOException
```

In the child class which of the following methods we can take?

Overriding ① public void m1(int i)

Overloading ② public static int m1(long l)

Overriding ③ public static void m1(int i)

Overriding ④ public void m1(int i) throws Exception

⑤ public static abstract void m1(double d)

→ CE: illegal combination of modifiers.

* Polymorphism :- One name but multiple forms is the concept of polymorphism.

eg:- method name is the same but we can apply for different types of arguments. (overloading).

abs(int)
abs(long)
abs(float)
} overloading.

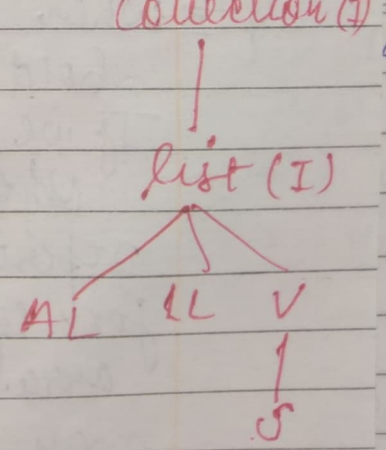
eg:- Method Signature is same but in parent class one type of implementn and in the child class another type of implementn.. [overriding].

```
class P
{
  marry()
  {
    sopln("dubalam");
  }
}
class C extends P
{
  marry()
  {
    sopln("3mu/muz/abc");
  }
}
```

Overriding

eg3:- Usels of parent reference to hold child Object is the concept of polymorphism.

```
List l = new AL();
         new LL();
         new stack();
         new vector();
```

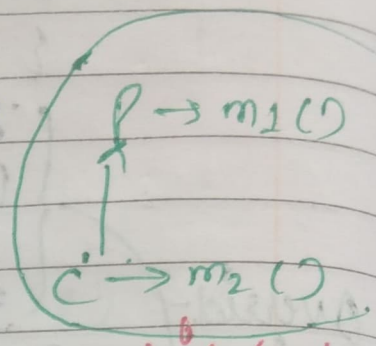


CLASSTIME Pg. No.
Date / /

Parent class reference can be used to hold child object, but by using that reference we can call only the methods available in parent class. and we can't call child specific methods.

But by

```
P p = new C();
p.m1();
p.m2(); X
```



CE: Cannot find symbol
Symbol: method m2()
Location: class P

But By using child object reference we can call both parent & child class methods.

```
C c = new C();
c.m1();
c.m2();
```

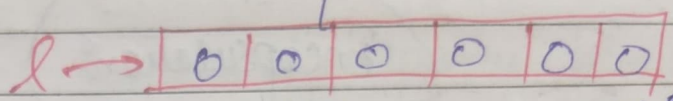
Q→ When we should go for parent reference to hold child object.

If we don't know exact Runtime type of object then we should go for parent reference.

for eg: The first element present in the arrayList can be any any type: it may be student object, customer, String, stringBuffer object, hence the

return type of get method is Object, which can hold any object.

```
Object o = l.get(0);
```



Heterogeneous Object

```
List l = m1();

public List m1()
{
    AL/LL/V/Stack
}
```

C c = new C(); Eg: AL l = new AL();	P p = new C(); Eg: List l = new AL();
(1) We can use this approach if we know exact runtime type of object.	1) We can use this approach if we don't know exact runtime type of object.
2) By using child reference we can call both parent class & child class methods. (this is the advantage of this approach)	2) By using parent reference we can call only methods available in parent class. and we can't call child specific methods. (this is the disadvantage of this approach.).
(3) We can use child reference to hold only particular child class object.	3) We can use parent reference

CLASSTIME | Pg. No.
Date / /

(This is the disadvantage of this approach) | to hold any child class object. (This is the advantage of this approach).

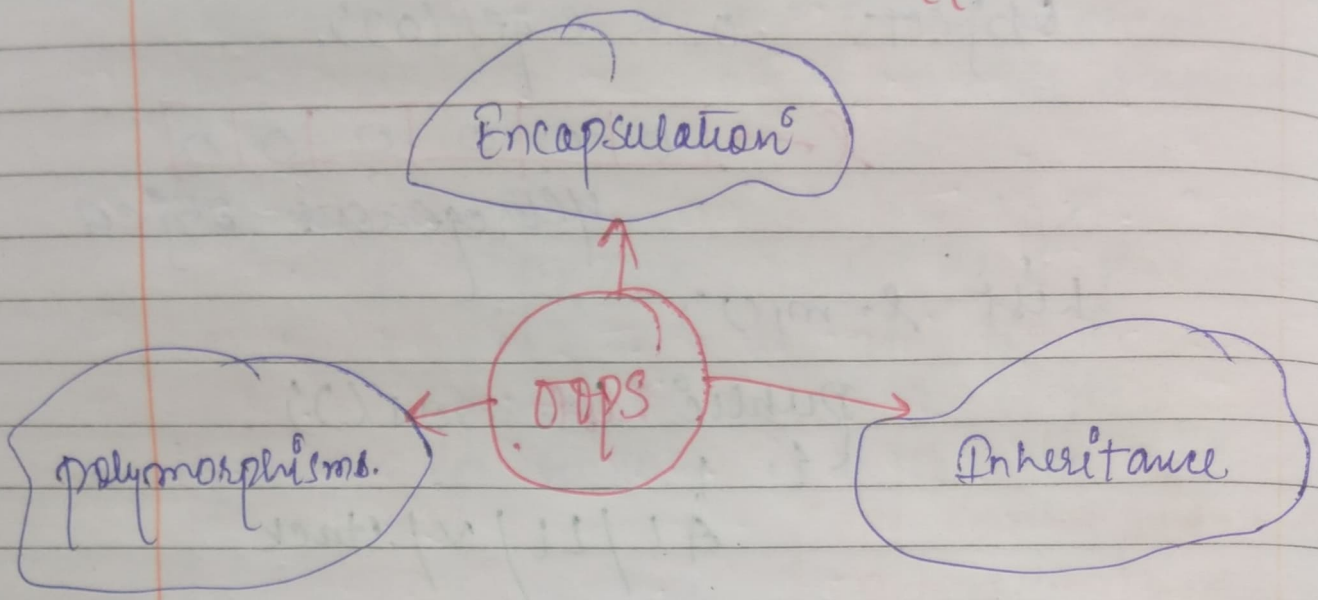


fig: 3 pillars of OOPS

Polymorphism.

Static polymorphism (OR) Compile-time polymorphism (OR) Early binding

Overloading
Method Hiding

Dynamic polymorp. (OR) Runtime polymorp. (OR) Late binding.

↓

Overriding

A BOY starts Love with the word Friendship but GIRL end Love with the same word Friend. word is same but attitude of effort called polymorp. hi

play less role at basic.

* Lecture 58 :- Coupling (Used in advance level like framework level, MVC, High Level Design, Struts, Hibernate, web & interprise application, designing level)

(1) * Coupling :- The degree of depency b/w the components is called Coupling.

If dependy is more then it is considered as tightly coupling. and If dependency is less then it is considered as Loosely Coupling.

eg :- ① class A
{
static int i = B.j;
}

② class B
{
static int j = C.K;
}

③ class C
{
static int K = D.m1();
}

④ class D
{
public static int m1()
{
return 10;
}
}

Tightly Coupled.

The above components are said to be tightly coupled with each other because dependency b/w the components is more.

Tightly coupling is not a good programming practice it has several serious disadvantages.

CLASSTIME | Pg. No.
Date / /

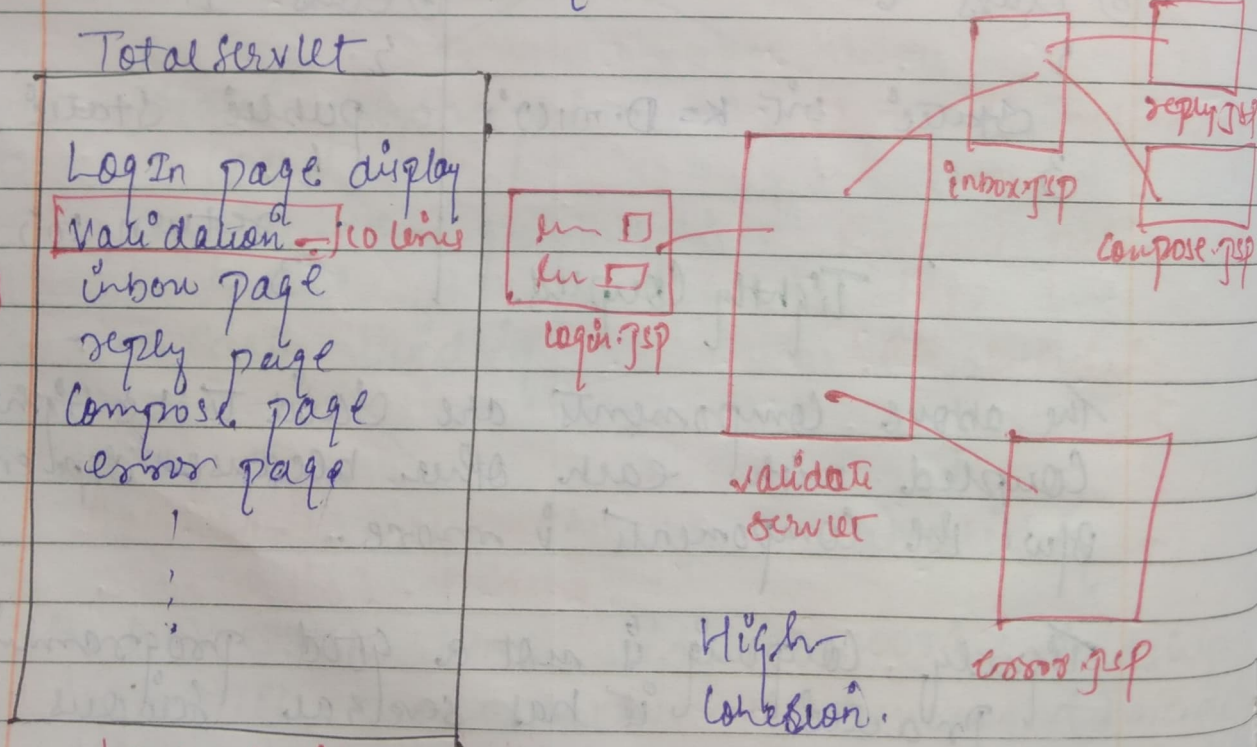
1) Without effecting remaining components we can't modify any component. And hence enhancement will become difficult.

2) It suppresses reusability.

3) It reduces maintainability of the applica.

4) Hence we have to maintain dependency b/w the components as less as possible. i.e. loosely coupling is a good programing practice.

(2) Cohesion :- for every component a clear well define functionality is required. defined then that component is said to be follow high cohesion.



High Cohesion is always a good programming practice because it has several advantages :-

(i) Without effecting remaining components, we can modify any component. Hence enhancement will become easy.

(ii) It promotes reusability of the code. (wherever validn is required we can reuse the same validate servlet). without rewriting.

(iii) It improves maintainability of the appl^{n}.

Note: Loosely coupling & high cohesion are good programming practices.

* Object Typecasting :- We can use parent reference to hold child object.

eg: Object o = new String("durga");

We can use interface reference to hold implemented class object.

eg: Runnable r = new Thread();

```
A b = (C) d;
```

A — class/interface name
b — Name of Reference variable
C — class/Interface name
d — Reference variable name.

Mantra :- (Compile time checking :-)

The type of 'd' and 'C' must have some relation either (child to parent, or parent to child or same child.) otherwise we will get CTE: inconvertible types

found: d type.
req: C

eg: Object O = new String("durga");
StringBuffer SB = (StringBuffer) O;

eg2:
String S = new String("durga");
StringBuffer sb = (StringBuffer) S;

CE: inconvertible types.
found: java.lang.String
req: java.lang.StringBuffer.

Memo 2 (Compile Time checking 2)

'C' must be either same or derived type of 'A'. Otherwise we will get CTE: incompatible types.
found: C
req: A.

eg-1: Object O = new String("durga");
StringBuffer SB = (StringBuffer) O;

eg-2: Object O = new String("durga");
StringBuffer SB = (String) O;

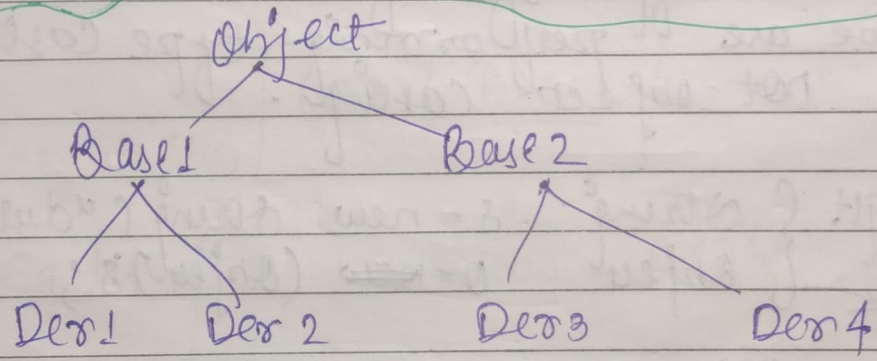
CE: incompatible types
found: S.L.String
req: T.L.SB

Mpoint 3 :- (Runtime checking (JVM)) :-
Runtime Object type of 'd' must be either same or derived type of 'c'. Otherwise we will get Runtime exception.
RE: ClassCastException.

eg-1 :- Object o = new String("durga");
StringBuffer sb = (StringBuffer) o;

RE: ClassCastException : java.lang.String cannot be cast to java.lang.StringBuffer.

eg-2 :- Object o = new String("durga"); ✓
Object o1 = (String) o;



Base2 b = new Der4();

① Object o = (Base2) b;
② Object o = (Base1) b; → CE: inconvertible types found: Base2 req: Base1
③ Object o = (Der3) b; → RE: CCE
④ Object o = (Der4) b;

CLASSTIME Pg. No.
Date / /

X (4) Base2 b1 = (Base1)b;

X (5) Base1 b1 = (Der1)b; → CE: Incompatible Types found: Der1 req: Base1

X (8) Base1 b1 = (Der1)b; →

{ CE: inconvertible types found: Base2 req: Der1. }

* lecture 59 :

Strictly speaking through type Casting we are not creating any new object.

for the existing object we are providing another type of reference variable i.e. we are performing type casting but not object casting.

eg1: String s = new String("durga");
Object o = (Object)s;

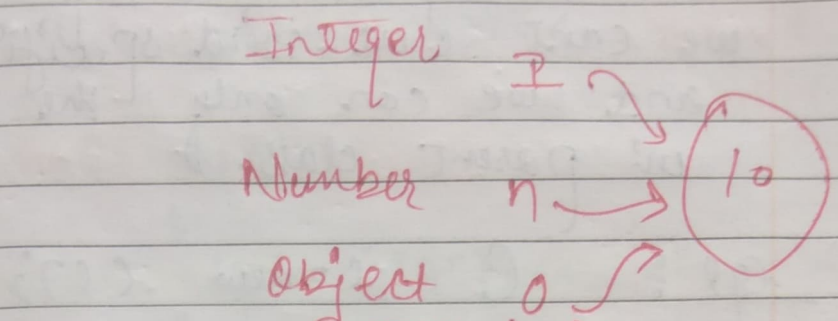
Object o = new String("durga");

String → s → durga
Object → o

eg2: Integer I = new Integer(10);
Number n = (Number)I;
Object o = (Object)n;

sopln(I==N); true

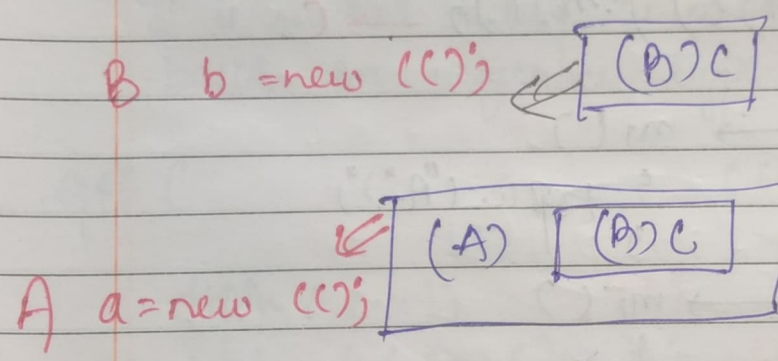
sopln (n==0); true

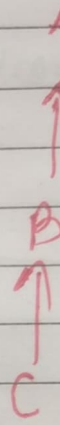


2 line → Number n = new Integer(10);

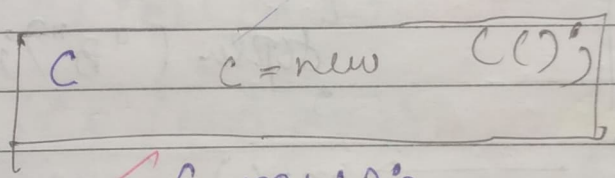
3 line → Object o = new Integer(10);

Note :- C C = new C();





eg-1 :-



✓ c.m1();
✓ c.m2();

P → m1(){}

C → m2(){}

✓ ((P)c).m1();

✗ ((P)c).m2();

P { P = new C();
P.m1(); }

P · P = new C();
P.m2();

CLASSTIME Pg. No.
Date / /

Reason: Parent reference can be used to hold child object, but by using that reference we can't call child specific methods. and we can only the methods available in parent class.

eg: C C = new C();

C.m1(); → C

((B)C).m1(); → C

((A)((B)C)).m1(); → C

```
A ---> m1()
|        {
|          sopln("A");
|        }
B ---> m1()
|        {
|          sopln("B");
|        }
C ---> m1()
         {
           sopln("C");
         }
```

It is overriding and method resolution is always based on runtime object

eg: C C = new C();

C.m1(); → C

((B)C).m1(); → B

((A)((B)C)).m1(); → A

A → static m1()
{
sopln("A");
}

B → static m1()
{
sopln("B");
}

C → static m1()
{
sopln("C");
}

method Hiding

It is method hiding & method resolution is always based on reference type.

eg:- C c=new C();

sopln(c.x); → 999
sopln(((B)c).x); → 888
sopln(((A)((B)c)).x); → 777

A — int x = 777;
B — int x = 888;
C — int x = 999;

Variable resolution is always based on reference type but not based on runtime object.

CLASSTIME Pg. No.
Date / /

★ Lecture 60 :- Static control flow :-

When ever we are executing a java class, the following sequence of steps will be executed. as the part of static control flow.

① Identification of static members from top to bottom. [1 to 6]

② Execution of static variable assignments & static blocks from top to bottom. [7 to 12]

③ Execution of main method. [13 to 15]

```
class Base
{
i=0[RIWO]  ① static int i=10 ⑦
j=0[RIWO]
i=10[RAW]  ② static
j=20[RAW]   {
              m1();  ⑧
              sopln("First static Block"); ⑩
            }
③ p s v main(String[] args)
  {
    m1();  ⑬
    sopln("main method"); ⑮
  }
④ p s v m1()
  {
    sopln(j);  ⑨, ⑭   j=0
  }
```

CLASSTIME Pg. No.
Date / /

```
}
⑤ static
{
    sopln ("second static Block");   ⑪
}
⑥ static int j = 20;                 ⑫
}
```

O/P:

Java Base ↓

0 — ⑨ → J=0.

First static Block. — 10

Second static Block. — ⑪

20

'main' method.

Inside a static block if we are trying to read a variable that read operation is called direct read.

If we are calling a method and within that method if we are trying to read a variable then that read operation is called indirect read.

Eg:

```
class Test
{
    static int i = 10;   → i=0 [RTWO]
    .
    static
    {
        m1();
```

Test identified by JVM

```
    sop(i);  → Direct read.
}
P  s v m1()
{
    sop(i);  → Indirect read.
}
}
```

If a variable is just identified by the JVM and original value not yet assigned then the variable is said to be in [Read Indirectly Write only state] RIWO.

If a variable is in Read Indirectly write only state. then that ~~read~~ operation we can't perform direct read but we can perform indirect read.

If we are trying to read directly then we will get CE: illegal forward reference.

eg:
```
class Test
{
① static int x=10; → ③

② static
  {
    sop(x); ← direct read
  }
}
```

O/P: 10

RE: NoSuchMethodError: main

CLASSTIME Pg. No.
Date / /

```
Class Test
{
(1) static
  {
(5)  sopln(x); — direct read.
  }
(2) static int x=10;
}
```

x=0 [RIWO]

(3) → CE: illegal forward reference.

```
Class Test
{
(1) static
  {
(4)  m1();
  }
(2) p s v m1()
  {
    sopln(x); — indirect read.
  }
(3) static int x=10;
}
```

x=0 [RIWO].

O/p: 0

RE: NoSuchMethodError: main.

Lecture 61 :- Static Block :-

Static blocks will be executed at the time of class loading hence, at the time of class loading if we want to perform any activity we have to define that inside

Static block.

eg:- At the time of Java class loading the corresponding native libraries should be loaded hence, we have to define this activity inside static block.

```
Class Test
{
  static
  {
    System.loadLibrary("native library Path");
  }
}
```

eg:- After loading every database drever class we have to register driver class with driver manager. But inside database drever class there is a static block. To perform this activity and we are not responsible to register explicitly.

```
eg:  class DBdrever
     {
       static
       {
         Register this Driver
         with drever manager.
       }
     }
```

CLASSTIME Pg. No.
Date / /

Note: Within a class we can declare any no of static blocks but all these static blocks will be executed from top to bottom.

Q1: Without writing is it possible to print some statements to the console?

Ans: Yes, By using static block.

```
class Test
{
  static
  {
    sopln("Hello I can print");
    System.exist(0);
  }
}
```

O/p: Hello I can print.

Q2: Without writing main method and static block is it possible to print some statement to the console?

Ans: Yes, of coure there are multiple ways.

```
class Test
{
static int x = m1();
public static int m1()
{
  System.s
  public static int m1()
  {
    System.out.println("Hello I can print");
```

```
        System.exist(0);
        return 10;
    }
}
```

→
```
class Test
{
    static Test t=new Test();
    {
        sopln("Hello I can print");
        System.exist(0);
    }
}
```

→
```
class Test
{
    static Test t = new Test();
    Test()
    {
        sopln("Hello I can print");
        System.exist(0);
    }
}
```

Note: From 1.7V onwards main method is mandatory to start a program execution. Hence, from 1.7V onwards with out writing main method it is impossible to print some statements to the console.

CLASSTIME Pg. No.
Date / /

* Static control flow in Parent to child relationship:

Whenever we are executing child class the following sequence of events will be executed automatically as the part of static control flow.

1. Identification of static members from parent to child. [1 to 11]
2. Execution of static variable assignments & static blocks from parent to child. [12 to 22]
3. Execution of only child class main method. [23-25]

```
i=0[R/W0]   class Base
j=0[R/W0]   {
x=0[R/W0]   static int i=10;   (12)
y=0[R/W0]   static   (2)
i=10[R/W]   {
j=20[R/W]   m1();   (13)
x=100[R/W]  System.out.println("Base static block");   (15)
y=100[R/W]  }
            (3) public static void main(String[] args)
                {
                m1();
                System.out.println("Base main");
                }
            (4) public void m1()
                {
                System.out.println(j);   (14)
                }
```

(5) static int j = 20; (16)

}

class Derived extends Base

{

(6) static int x = 100; (17)

(7) static

{

m2(); (18)

sopln("Derived First static Block"); (20)

}

(8) p s v main(String[] args)

{

m2(); (23)

sopln("Derived main"); (25)

}

(9) p s v m2()

{

sopln(y); (19), (24)

}

(10) static

{

sopln("Derived Second static Block"); (21)

}

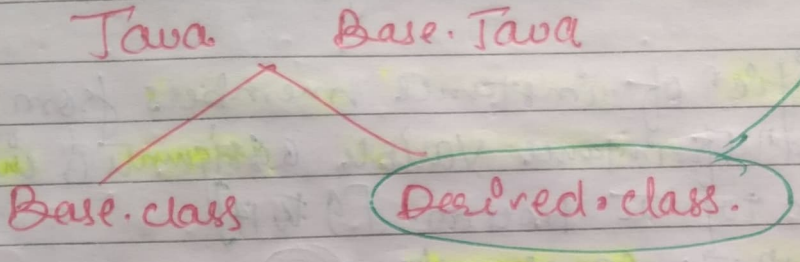
(11) static int y = 200; (22)

}

CLASSTIME Pg. No.
Date / /

Java Derived ↓

O/P:-
0
Base. Static Block
0
Derived. First Static Block
Derived second. Static Block
200
Derived. main.



Java Base ↓ → for execution of parent.
0
Base. Static Block
20
Base. main.

Note:- Whenever we are loading child class automatically parent class will be loaded, but whenever we are loading parent class child class won't be loaded. [Because parent class members by default availble to the child class whereas child class members by default won't available to the parent].

CLASSTIME Pg. No.
Date / /

Lecture 62 : Instance control flow :

Whenever we are executing a Java class. first static control flow will be executed. In the static control flow if we are creating an object the following sequence of events will be executed as a part of instance control flow.

[3 — 8]

① Identific^n of instance members from top to bottom
② Execut^n of instance varible assigmnts & instnce blocks from top to bottm. [9 to 14]
③ Execut^n of constructrs. (15)

```
class Test
{
  ③ int i=10; (9)
j=0[RIWO]    { ④
j=0[RIWO]      m1(); (10)
i=10[R&W]      Sopln ("First instance block"); (12)
j=20 [R&W]   }
  ⑤ Test()
    {
      Sopln ("constructor"); (15)
    }
  ① p s v main (String[] args)
    {
  ② Test t = new Test(); — (line 1)
      Sopln ("main"); (16)
    }
```

```
(6) public void m1();
    {
        System.out.println(j); (11)
    }
(7) {
        System.out.println("second Instance block."); (13)
    }
(8) int j = 20; (14)
}
```

Java Test 4

0
First Instance block
second Instance block
Constructor
main.

Note: If we comment line 1 then output is: main.

Note: Static control flow is one time activity which will be performed at the time of class loading.
But i.e instance control flow is not one time activity and it will be performed for every object creation.

Object creation is the most costly operation if there is no specific requirement then it is not recommended to create object.

CLASSTIME Pg. No.
Date / /

* Instance control flow in parent to child relationship :

Whenever we are creating child class object the following sequence of events will be performed automatically as a part of instance control flow.

1. Identification of instance members from parent to child. [4 to 14]
2. Execution of instance variable assignments [15-19] and instance blocks only in parent class.
3. Execution of parent constructor. (20)
4. Execution of instance variable assignments and instance blocks in child class. [21 to 26]
5. Execution of child constructor. (27)

```
class Parent
{
    int i=10;                          (4) (5)
    {                                  (6)
        m1();                          (16)
        sopln("Parent Instance block"); (18)
    }
    Parent()                           (6)
    {
        sopln("Parent Constructor");   (20)
    }
    p s v main(String[] args)          (1)
    {
        Parent P = new Parent();
```

i=0 [RI WO]
j=0 [RI WO]
x=0 [RI WO]
y=0 [RI WO]
i=10 [RS WI]
j=20 [RS WI]
m=100 [RS WI]
y=200 [RS WI]

```
    soPln("Parent main");
}
(7) public void m1()
{
    soPln(j);  (17)
}
(8) int j = 20;  (19)
}

class child extends Parent
{
(9) int x=100;  (21)
(10) {
    m2();  (22)
    soPln("CFIB");  (24)
}
(11) child()
{
    soPln("child constructor");  (27)
}
(2) p s v main(String[] args)
{
(3) child c=new child();
(28) soPln("child main");
}
(12) public void m2()
{
    soPln(y);  (23)
}
(13) {
```

CLASSTIME | Pg. No.
Date / /

```
{ soplm ("CSIB"); (25)
3
(24) int y = 200; (26)
3
```

Java → Parent Java
Parent class
child class

Java child ↓
Parent Instance Block
Parent Constructor
CFIB
CSIB
child constructor
child main.

* Lecture 63 :- Instance & In static block :-

```
class Test
{
  {
   soplm ("FIB");
  }
  static
  {
   soplm ("FSB");
  }
  Test()
```

```
{
sopln ("constructor");
}
P S V main (String[] args)
{
Test t1 = new Test();
sopln ("main");
Test t2 = new Test();
}
Static
{
sopln ("SSB");
}
{
sopln ("SIB");
}
}
```

op:

FSB
SSB
FIB
SIB
Constructor
main
FIB
SIB
constructor

- 1 Time

CLASSTIME Pg. No.
Date / /

eg2:-
```
public class Initialization
{
    private static String m1(String msg)
    {
        System.out.println(msg);
        return msg;
    }
    public Initialization()
    {
        m = m1("1");   ⑥
    }
    {
        ④ m = m1("2");
    }
    String m = m1("3");   ⑤
    ② public static void main(String[] args)
    {
        ③ Object o = new Initialization();
    }
}
```

Op: 2
3
1

m = ~~null~~
~~2~~ ~~3~~ 1

eg3:-
```
public class Initialization2
{
    private static String m1(String msg)
    {
        System.out.println(msg);
        return msg;
    }
}
```

```
static String m = m1("1");
{
  m = m1("2");
}
static
{
  m = m1("3");
}
psvm(String[] args)
{
  Object o = new Initialization();
}
}
```

O/P: 1
3
2

m = ~~null~~
1
~~3~~
2

Note: From static area we can't access instance members ~~because while~~ directly, because while executing static area JVM may not identify instance members.

```
class Test
{
  int x = 10;
  psvm(String[] args)
  {
    sop(x);
  }
}
```

→ CE: non-static variable x cannot referenced from static content.

```
Test t = new Test();
```

CLASSTIME | Pg. No.
Date / /

sop(t.x); ✓ [o/p :- 10]

3 3

Imp

Ques: In how many ways we can create an Object in° Java or in° how many ways we can get Object in° Java.?

① By using new operator :-

```
Test t = new Test();
```

② By using newInstance() method:-

```
Test t = (Test) class.forName("Test").
                 newInstance();
```

③ By using Factory method:-

```
Runtime r = Runtime.getRuntime();
DateFormat df = DateFormat.getInstance
                ();
```

④ By using clone() method:

```
Test t1 = new Test();
Test t2 = (Test) t1.clone();
```

⑤ By Using Deserialization:

FileInputStream fis = new FIS("abc.ser");

ObjectInputStream ois = new OIS(fis);

Dog d2 = (Dog) ois.readObject();

Lecture 64:- Constructor :-

Once we creates an object, compulsory we should perform initialization then only the object is in a position to respond properly.

Whenever we are creating an object some piece of the code will be executed, automatically to perform initialization of the object. this piece of the code is nothing but constructor. Hence the main purpose of constructor is to perform initialization of an object.

Eg:-

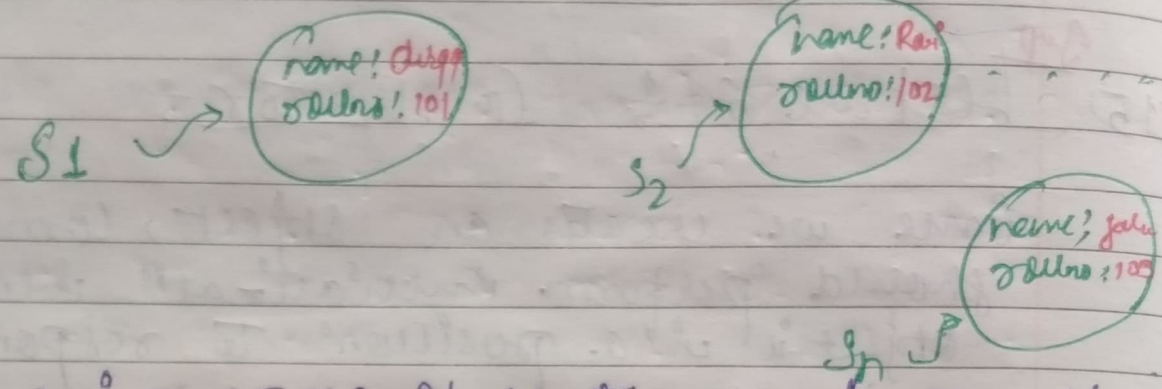
```
class Student
{
    String name;
    int rollno;

    Student(String name, int rollno)
    {
        this.name = name;
        this.rollno = rollno;
    }
```

Constructor

CLASSTIME Pg. No.
Date / /

```
psvm(String[] args)
{
    Student s1 = new Student("durga",101);
    Student s2 = new Student("Ravi",102);
    :
    3 :
}
```



Note:-

The main purpose of constructor is to perform initialization of an object, but not to create object.

Que:- Difference b/w Constructor & instance block:-

The main purpose of constructor is to perform initialization of an object.
But other than initialization if we want to perform any activity for every object creation. then we should go for instance block. (like updating one entry in the database for every object creation. or incrementing count value for every object creation etc.

Both constructor & instance block have their own purposes. and replacing one concept with another concept may not work

always.

Both constructor and instance block will be executed for every object creation, but instance block first, constructor followed by constructor next.

Demo program to print no. of objects created for a class.

```
class Test
{
  static int count = 0;
  {
    count++;
  }
  Test()
  {
  }
  Test (int i)
  {
  }
  Test (double d)
  {
  }
p s v main (String[] args)
{
    Test t1 = new Test ();
    Test t2 = new Test (10);
    Test t3 = new Test (10.3);
    System.out.println ("The no of objects created: " + count);
  }
}
```

CLASSTIME Pg. No.
Date / /

* Rules of writing constructors :-

(i) Name of the class and name of the constructor must be matched.

(ii) Return Type of concept not applicable for constructor even void also.

(iii) By mistake if we are trying to declare return type for the constructor. then we won't get any CTE because compiler treat it as a method.

```
class Test
{
    void Test()
    {
    }
}
```

It is a method but not constructor.

Hence it is legal (but worst) to have a method whose name is exactly same as class name.

eg:-
```
class Test
{
    void Test()
    {
        sopln("method but not constructor");
    }
    p s v main(String[] args)
    {
```

```
Test t=new Test();
t.Test();
}
}
```

(iv) The only applicable modifiers for constructors are public, private, protected, default. If we are trying to use any other modifier we will get CTE.

```
class Test
{
    static Test()
    {
    }
}
```

CE: modifier static not allowed here.

* default constructor :-

(1) Compiler is responsible to generate default constructor (but not JVM).

(2) If we are not writing any constructor then only compiler will generate default constructor. i.e. if we are writing at least one constructor then compiler won't generate default constructor. hence every class in Java can contain constructor it may be default constructor generated by compiler or customized constructor.

explicitly provided by compiler but not both simultaneously.

```
class Test
{
    Test()
    {
    }
}
```

```
class Test
{
    Test(int i)
    {
    }
}
```

Lecture 65 : 8/25/22 12:25PM

Prototype of default constructor :

(1) It is always no-arg constructor.
(2) The access modifier of default constructor is exactly same as access modifier of class. [This rule is applicable only for public & default].
(3) It contains only one line. `super();`
(4) It is a no-arg call to super class constructor.

```
class Test
{
    Test()
    {
        super();
    }
}
```

CLASSTIME Pg. No.
Date / /

Programmer Code	Compiler Generated Code

class Test
{
}
}
	class Test
	{
	Test()
	{
	super();
	}
	}

public class Test
{
}
}
	public class Test
	{
	public Test()
	{
	super();
	}
	}

public class Test
{
void Test()
{
}
}
}
	public class Test
	{
	public Test()
	{
	super();
	}
	} → void Test()
	{
	}

class Test
{
Test()
{
}
}
	class Test
	{
	Test()
	{
	super(); But super I. or this();
	}
	}

class Test
{
Test(int i)
{
super();
}
}
	class Test
	{
	Test(int i)
	{
	super();
	}
	}

```
class Test
{
  Test()
  {
    this(10);
  }
  Test(int i)
  {
  }
}
```

```
class Test
{
  Test()
  {
    this(10);
  }
  Test(int i)
  {
    super();
  }
}
```

The first line inside every constructor should be either super or this. and if we are not writing anything then, compiler will always place [super();].

Case I: We can take super() or this() only in first line of constructor. If we are trying to take anywhere else we will get CTE.

eg:

```
class Test
{
  Test()
  {
    Sopln("constructor");
    super();
  }
}
```

CE: Call to super must be first statement in constructor.

CLASSTIME | Pg. No. | Date / /

Case2: Within the constructor we can take either super or this() but not both simultaneously.

```
class Test()
{
   Test()
   {
     super();
     this();
   }
}
```

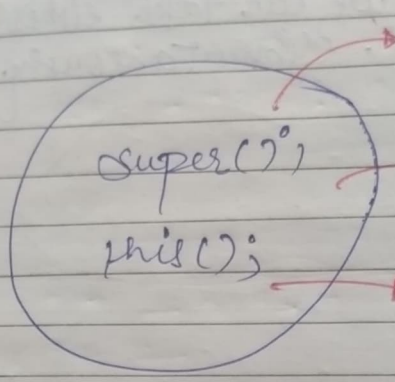
CE: Call to this must be first statement in constructor.

Case3: We can use super() or this() only inside a constructor, if we are trying to use outside of constructor we will get CTE.

```
class Test
{
   public void m1()
   {
      super();
      sop("Hello");
   }
}
```

CE: Call to super must be first statement in constructor

i.e. we can call a constructor directly from another constructor only.



→ We can use only in constructors

→ only in first line

→ Only one but not both simultaneously.

super(), this()	super, this
(1) These are constructor calls to call super class and current class constructors.	(1) These are keywords to refer superclass and current class instance members.
(2) We can use only in constructors as first line.	(2) We can use anywhere except static area.
(3) We can use only once in constor.	(3) We can use any no of times.

(2) eg: class Test

```
{
    P S V main(String[] args)
    {
        sopln(super.hashCode());
    }
}
```

CLASSTIME | Pg. No. | Date / /

If non-static variable super cannot be referenced from a static context.

* Just for differtiat" of keywords & constructor calls.

```
class P
{
  int x=100;
}
class C extends P
{
   int x=200;
   public void m1()
   {
      sopln (this.x); 200
      sopln (super.x); 100
   }
}
```

→ Both are keywords.

1:24 PM : Time
08/25/22 : DATE

Two lectures left to complete OOPS concepts - will in next copy / Note.

Date 8/25/22
Page 01

OOPS continue >>>

Lecture 66 :- Overloaded Constructors :-

Within a class we can declare multiple constructors and all these constructors having same name but different type of arguments. Hence overloaded all these constructors are considered as overloaded constructors. Hence, Overloading concepts are applicable for constructors.

eg:-

```
class Test
{
    Test()
    {
        this(10);
        sopln("no-arg");
    }
    Test(int i)
    {
        this(10.5);
        sopln("int-arg");
    }
    Test(double d)
    {
        sopln("durga-arg");
    }
}
```

Overloaded constructors

```
P s v main(String[] args)
{
    Test t1 = new Test();   - double-arg
                              int-arg
                              no-arg

    Test t2 = new Test(10); - double-arg
                              int-arg.

    Test t3 = new Test(10.5); - double-arg

    Test t4 = new Test(10L); double-arg.

}
```

For constructors inheritance and overriding concept are not applicable. but overloading concept is applicable.

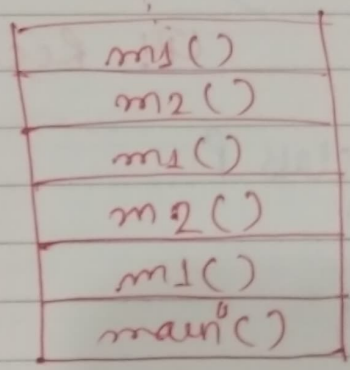
Every class in Java including abstract class can contain constructors, but interface cannot contain constructor.

class Test	abstract class Test	interface Test
{	{	{
Test()	Test()	Test()
{	{	{
}	}	}
} ✓	} ✓	} ✗

Case 1: Recursive method call is a runtime exception
RE: StackOverflowError.

But in our program if there is a chance of recursive constructor invocation then the code won't compile and we will get CTE: Recursive constructor invocation.

```
class Test
{
    p s v m1()
    {
        m2();
    }
    p s v m2()
    {
        m1();
    }
    p s v main(String[] args)
    {
        m1(); ——①
        sopln("Hello");
    }
}
```



RE: StackOverflowError

```
class Test
{
    Test()
    {
        this(10);
    }
    Test(int i)
    {
```

```
this();
}
p s v main(String[] args)
{
    sopln("Hello");
}
}
```

CE: Recursive constructor invocation

Case 2:

```
class P
{
P()
{
super();
}
class C extends P
{
C()
{
super();
}
}
```

✓

```
class P
{
    P()
    {super();
    }
}
class C extends P
{
    C()
    {
    }super();
}
```

✓

```
class P
{
    P(int i)
    {
        super();
    }
}
class C extends P
{
    C()
    {
    }super();
}
```

CE: Cannot find symbol
Symbol: Constructor P()
Location: class P.

✗

Note 1: If parent class contains any argument constructor then while writing child classes, we have to take special care w.r.t constructors.

* 2: Whenever we are writing any argument constructor it is highly recommended to write no arg constructor also.

Exp 2:

Case 3:

```
class P
{
    P() throws IOException
    {
    }
}
class C extends P
{
    C()
    {
        super();
    }
}
```

CE: unreported exception
Java.io.IOException
in default constructor.

```
class P
{
    P() throws IOException
    {
    }
}
```

```
class C extends P
{
  C() throws IOException/Exception/Throwable
  {
    super();
  }
}
```

✓

Note: If parent class constructor throws any checked exception compulsory child class constructor should throw the same checked exception or its parents otherwise the code won't compile.

Q- Which of the following is valid?

i) The main purpose of Constructor is to create an object. false

ii) The main purpose of Constructor is to perform initialization of an object. True ✓

iii) The name of the constructor need not be same as class name. false ✗

iv) Return type concept applicable for constructors but only void. ✗ false

v) We can apply any modifier for constructor ✗

vi) Default Constructor generated by JVM. ✗

vii) Compiler is responsible to generate default constructor. ✓

viii) Compiler will always generate default constructor - false

ix) If we are not writing no-arg constructor, then compiler will generate default Constructor. ✗ false.

(x) Every no-arg constructor is always default constructor. — false

xi) default constructor is always no-arg constructor. ✓

xii) The first line inside every constructor should be either super(); or this(); if we are not writing anything then compiler will generate this. — false

(xiii) for constructor both Overloading and Overriding concepts are applicable. — false

xiv) for constructors inheritance concept applicable but not overriding. — false

xv) Only concrete classes can contain constror but abstract classes cannot. ✗

(xvi) Interface can contain constructors. ✗

xvii) Recursive constructor invocation is a runtime exception. — false

xviii) If parent class constructor throws some checked exception then compulsory child class constructor should throws the same checked exception or its child. — false

~~(xix)~~

Lecture 67 : Singleton classes :

for any Java class if we are allow to create only one object such type of class is called Singleton class. eg : Runtime, Business Delegate, ServiceLocator etc.

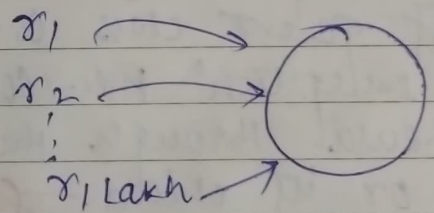
Advantage of Singleton class :

If several people have same requirement then

it is not recommended to create a seperate object for every requirement.
We have to create only One object and we can reuse the same object for every similar requirements. so that performance and memory utilizat" will be improved.

This is the central idea of Singleton classes.

Eg:- Runtime r1 = Runtime.getRuntime();
Runtime r2 = Runtime.getRuntime();
:
Rintime r1lakh = Runtime.getRuntime();



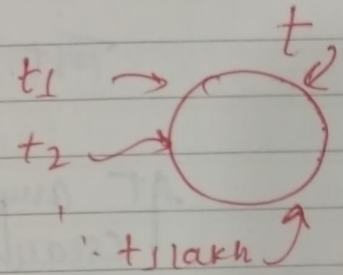
* How to create our own Singleton classes:-
We can create our own Singleton classes for this we have to use private constructor. and private static variable. and public factory method.

Approach 1:-

```
Class Test
{
```

```
private static Test t=new Test();
private Test()
{
}
public static Test getTime()
{
  return t;
}
}
```

```
Test t1 = Test.getTest();
Test t2 = Test.getTest();
  :
Test t1lakh = Test.getTest();
```



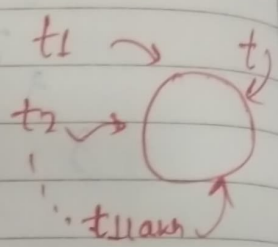
NOTE: Runtime class is internally implemented by using this approach.

* Approach 2:

```
class Test
{
  private static Test t=null;
  private Test()
  {
  }
  public static Test getTest()
  {
    if(t==null)
```

```
    {
      t=new Test();
    }
    return t;
  }
}
```

```
Test t1 = Test.getTest();
Test t2 = Test.getTest();
 .
 .
 .
Test t1lakh = Test.getTest();
```



At any point of time for test class we can create only one object, hence Test class is singleton class.

Q:- class is not final but we are not allowed to create child classes, how it is possible?

By declaring every constructor as private we can restrict child class creation.

```
class P
{
  private P()
  {
  }
}
```

for the above class it is impossible to create child class.

Ends of OOPs